|  |  |
| --- | --- |
|  |  |

EPTF AppLib MBT for TitanSim, User Guide

Contents

[1 Introduction 2](#_Toc272742595)

[1.1 Revision history 2](#_Toc272742596)

[1.2 About this Document 2](#_Toc272742597)

[1.2.1 How to Read this Document 2](#_Toc272742598)

[1.2.2 Presumed Knowledge 2](#_Toc272742599)

[1.2.3 References 2](#_Toc272742600)

[1.2.4 Abbreviations 3](#_Toc272742601)

[1.2.5 Terminology 3](#_Toc272742602)

[1.3 System Requirements 3](#_Toc272742603)

[2 Model Based Testing of Telecommunication Software 4](#_Toc272742604)

[2.1 Model Based Testing Technology 4](#_Toc272742605)

[2.2 TitanSim as MBT Test Harness 6](#_Toc272742606)

[3 MBT Tools 8](#_Toc272742607)

[3.1 Conformiq Qtronic 8](#_Toc272742608)

[3.2 Spec Explorer ‎[7] 10](#_Toc272742609)

[4 The MBT Application Library 14](#_Toc272742610)

[4.1 Overview 14](#_Toc272742611)

[4.2 Simple MBT Applib Demo 15](#_Toc272742612)

[4.2.1 Test arrangement 15](#_Toc272742613)

[4.2.2 Tester Behavior 16](#_Toc272742614)

[4.2.3 Test Harness Behavior 17](#_Toc272742615)

[4.2.4 Testcase Execution Example 18](#_Toc272742616)

[4.3 MBT Applib Demo with Qtronic using User Defined Messages 19](#_Toc272742617)

[4.3.1 SUT: SIP Phone 19](#_Toc272742618)

[4.3.2 Demo Files 20](#_Toc272742619)

[4.3.3 Test Arrangement Overview 21](#_Toc272742620)

[4.3.4 Tester Code Examples 21](#_Toc272742621)

[4.3.5 Test Harness Code Examples 23](#_Toc272742622)

[4.3.6 Workflow 25](#_Toc272742623)

[4.3.7 TTCNScripter settings 26](#_Toc272742624)

[4.4 MBT Applib Demo with Qtronic using Applib PDUs 28](#_Toc272742625)

[4.4.1 Demo Files 28](#_Toc272742626)

[4.4.2 Test Arrangement Overview 29](#_Toc272742627)

[4.4.3 Modeling with TitanSim API 30](#_Toc272742628)

[4.5 Demo with Spec Explorer using User Defined Messages 31](#_Toc272742629)

[4.5.1 Generating TTCN code in Spec Explorer 32](#_Toc272742630)

[4.5.2 Export2TTCN.cs 32](#_Toc272742631)

[4.5.3 The procedure of creating a new project 33](#_Toc272742632)

[4.5.4 FSM implementation techniques 34](#_Toc272742633)

[4.5.5 Demo files 35](#_Toc272742634)

[4.5.6 Exploration 37](#_Toc272742635)

# Introduction

## Revision history

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Rev | Characteristics | Prepared |
| 2009-12-16 | PA1 | First draft version | EANTWUH |
| 2010-04-14 | PA2 | Qtronic Demo with Applib PDUs updated. | EANTWUH |
| 2010-09-20 | A | Final revision | EANTWUH |
|  |  |  |  |

## About this Document

### How to Read this Document

This is the User Guide for the MBT application library of the Ericsson Performance Test Framework (TitanSim). The application library is developed for the TTCN-3‎[1] Toolset with TITAN ‎[2]. This document should be read together with Product Revision Information ‎[3] and Function Specification ‎[4]. Additionally, the TitanSim Core Load Library ‎[5] should be consulted for understanding the core functionalities of the TitanSim used by the application library.

### Presumed Knowledge

To use this applib the knowledge of the TTCN-3 language ‎[1] is essential.

### References

1. ETSI ES 201 873-1 v3.2.1 (2007-02)  
   The Testing and Test Control Notation version 3. Part 1: Core Language
2. 1/198 17-CRL 113 200 Uen  
   User Guide for the TITAN TTCN-3 Test Executor
3. 109 21-CNL 113 659-1 U  
   EPTF Applib MBT for TitanSim, Product Revision Information
4. 155 17-CNL 113 659 Uen  
   EPTF Applib MBT for TitanSim, Function Specification
5. 109 21-CNL 113 512-2 Uen   
   TitanSim CLL for TTCN-3 toolset with TITAN, Product Revision Information
6. EPTF MBT Application Library for TTCN-3 Toolset with TITAN, Reference Guide  
   <http://ttcn.ericsson.se/products/libraries.shtml>  
   Alternatively, please consult the doc/apidoc directory of the released application library.
7. Spec Explorer 2010 Help
8. Conformiq Qtronic  
   www.conformiq.com

### Abbreviations

Applib Application Library

CLL Core Load Library

EPTF Ericsson Performance Test Framework

FSM Finite State Machine

MBT Model Based Testing

MSC Message Sequence Chart

TTCN-3 Test and Test Control Notation version 3

SUT System Under Test

TitanSim New synonym for the EPTF framework

### Terminology

*TitanSim Core (Load) Library(CLL)* is that part of the TitanSim software that is totally project independent. (I.e., which is not protocol-, or application-dependent). The TitanSim CLL is to be supplied and supported by the TCC organization. Any TitanSim CLL development is to be funded centrally by Ericsson.

*TitanSim Appliaction Library* is the application-specific part of the TitanSim software.It provides load generation functionalities belonging to specific protocols or products.

## System Requirements

Application libraries are a set of TTCN-3 source code files that can be used as part of TTCN-3 test suites only. Hence, application libraries alone do not put specific requirements on the system used. However in order to compile and execute a TTCN-3 test suite using the set of application libraries the following system requirements must be satisfied:

* TITAN TTCN-3 Test Executor ‎[2] version 1.7.pl2 installed.

# Model Based Testing of Telecommunication Software

Testing takes a vital role in the software development process because creating software is an error-prone activity. Software faults occur through the following processes: a programmer makes an error (mistake), which results in a defect (fault, bug) in the software source code. If this defect is executed, in certain situations the system will produce wrong results, causing a failure. In order to avoid the consequences of errors, we must check the product in some systematic way.

The definition of testing, from the IEEE Software Engineering Body of Knowledge (SWEBOK 2004) describes the top-level goals of testing: Software testing consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior.

An important criterion that applies to telecommunications software is compatibility with systems from different vendors. This is usually achieved by the means of unambiguous specifications defined by standardization organizations. Manufacturers develop actual products according to these specifications to ensure compatibility. Conformance testing provides the means to check whether systems operate correctly according to the standard. It is crucial to create adequate test sets to minimize the time spent with testing without sacrificing reliability.

The test development process itself involves significant resources: it is very time consuming and requires the manual effort of many well-trained developers. Therefore, its automation is an important challenge.

## Model Based Testing Technology

Model-based testing is software testing in which test cases are derived in whole or in part from a model that describes some (usually functional) aspects of the system under test (SUT).

Model-based testing is the automation of black-box test design. A model based testing tool uses various test generation algorithms and strategies to generate tests from a behavioral model of the SUT.



Figure 1 Model Based Testing

The model is usually an abstract, partial presentation of the system under test's desired behavior. The model must be concise and precise: concise so that it does not take too long to write and so that it is easy to validate with respect to the requirements but precise enough to describe the behavior that is to be tested.

Test cases (including test data and oracles) can be automatically generated from the model using a model-based testing tool. The test engineer can also control the tool to focus the testing effort and manage the number of tests that are generated. The test cases derived from this model are functional tests on the same level of abstraction as the model. These test cases are collectively known as the abstract test suite. The abstract test suite cannot be directly executed against the system under test because it is on the wrong level of abstraction.

The tests produced from the model are abstract tests, so they must be transformed into executable tests. Therefore an executable test suite must be derived from the abstract test suite that can communicate with the system under test. This is done by mapping the abstract test cases to concrete test cases suitable for execution. This also requires some input from the test engineer, but most model-based testing tools provide assistance with this process.



Figure 2 General Workflow with MBT

## TitanSim as MBT Test Harness

The output of the MBT tools is an abstract test suite, where each testcase is usually described with a high-level MSC which contains abstract messages to be sent and to be expected as answers. This test suite cannot be executed directly against the SUT. Therefore usually a test harness is developed that can take the abstract messages and turn them into a real life messages with the help of templates and some algorithms. Naturally, the test harness must be able to convert real-life messages to abstract high level messages as well.



Figure 3 Test Harness Role

In Figure 3 an executable test suite with a test harness is depicted. The Abstract Test Implementation is in fact an MSC Executor. It is connected with the test harness and can send/receive abstract messages to/from the test harness. It can also create verdicts whether the MSC execution was successful or it failed.

The test harness handles this upper level API, so it can translate the abstract messages to real life messages and vice versa. To be able to do this, the test harness must implement some protocol specific functions. For example:

* Transport protocol handling
* Timeout/retransmission handling
* Checksum calculation
* Unique id generation
* …

In fact it must be able to handle all those protocol details, which were left out in the abstract model. Therefore development of such a framework is a difficult and laborious process.

Since developing a test harness is expensive one might think it would be a good idea to build a generic test harness, which can then be reused for several project. The development of such a generic test harness should be started by defining the higher-level interface of the test harness and the structure of the abstract messages. Since this interface heavily depends on the model, which in turn is very dependent on the test purpose the model was developed for, it is very hard to come up with a good interface that can be reused with as few limitations as possible.

One solution can be to re-use the application libraries that were developed for the TitanSim framework. These libraries have high level abstract interfaces. They can be extended with user code and they implement a lot of protocol specific functionality. In the following section we describe how these libraries can be used for the MBT approach.

# MBT Tools

## Conformiq Qtronic

Conformiq Qtronic (CQ) ‎[8] is a commercial MBT tool by Conformiq. It provides both online and offline MBT. CQ works on Windows and Linux operating systems. CQ is a general-purpose MBT tool. Thus, the models and test execution techniques and algorithms are not tied to any specific domain or platform. Qtronic provides its own components for modeling and test execution, but it can be integrated with external tools.

CQ has its own modeling tool but it also accepts inputting of models. Qtronic supports multi-threaded concurrent models and testing of non-deterministic systems in online mode. The modeling tool provided by CQ is the Qtronic Modeler and it uses Qtronic Modeling Language (QML). QML means UML statechart extended with java or C# code. Qtronic supports also CQλ and any UML2.0 models as input. CQλ is a variant of LISP. UML2.0 can be used for importing models from third party modeling tools. UML2.0 has to be saved in XMI format before importing. All of these can all be extended with java or with C# in the same manner as with Qtronic Modeler.

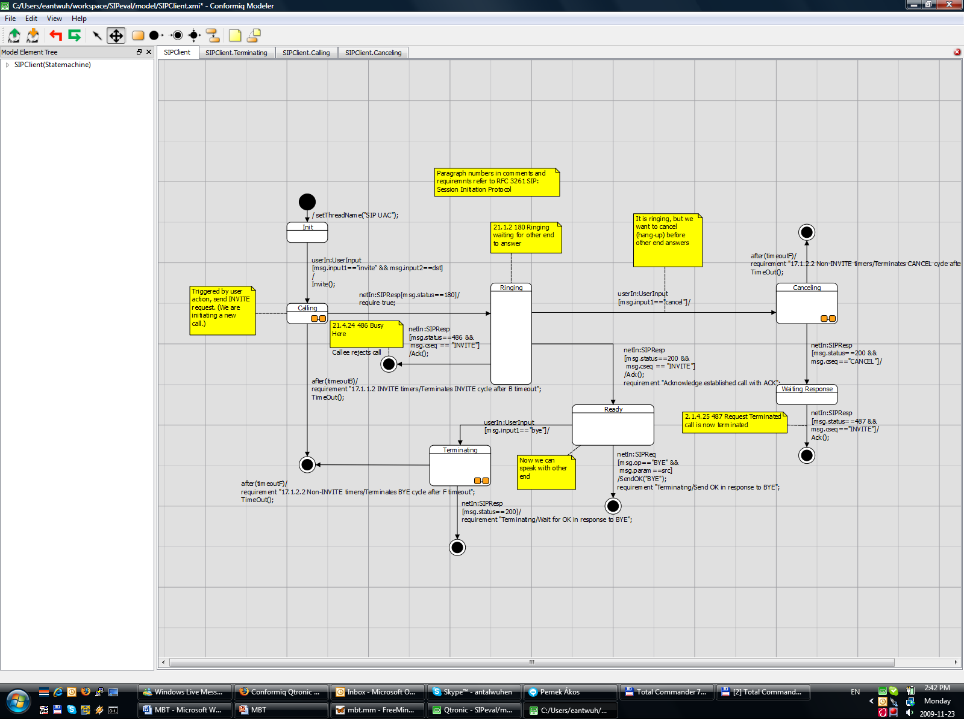


Figure 4 Qtronic Modeler

Qtronic provides nine sophisticated coverage criteria, which provide good test generation guiding possibilities. In offline generation, Qtronic is able to limit search tree depth and maximum delay for response. In similar fashion to LTD, Qtronic also provides test generation based on specification requirements, which are interpreted and described in the model. CQ also provides manually created use cases for test generation guidance. Qtronic providing coverage criteria are state coverage, transition coverage, 2-transition coverage, implicit consumption, boundary value analysis, branch coverage, method coverage, statement coverage, atomic condition coverage.

In Online, perspective mode the user can choose one of three alternative walking techniques: random, Markov Chain or coverage guided. The Markov Chain algorithm does not promote the same route again which means a wider scope of walking. The coverage guided walking technique focuses on covering selected coverage criteria. The coverage-guided technique is very useful when the testing time is short. In online mode, the user can also define the maximum latency time. Test execution can optionally be paused automatically, stopped when all coverage criteria are fulfilled, or stopped after a single test run.

In offline mode the user is able to choose look ahead depth and maximum delay time. Look ahead depth controls the amount of CPU time used for planning the test scripts. Maximum delay signifies response waiting time after the sending of input. The offline mode also makes it possible to minimize the size of the test sets, or to generate only finalized test sets.

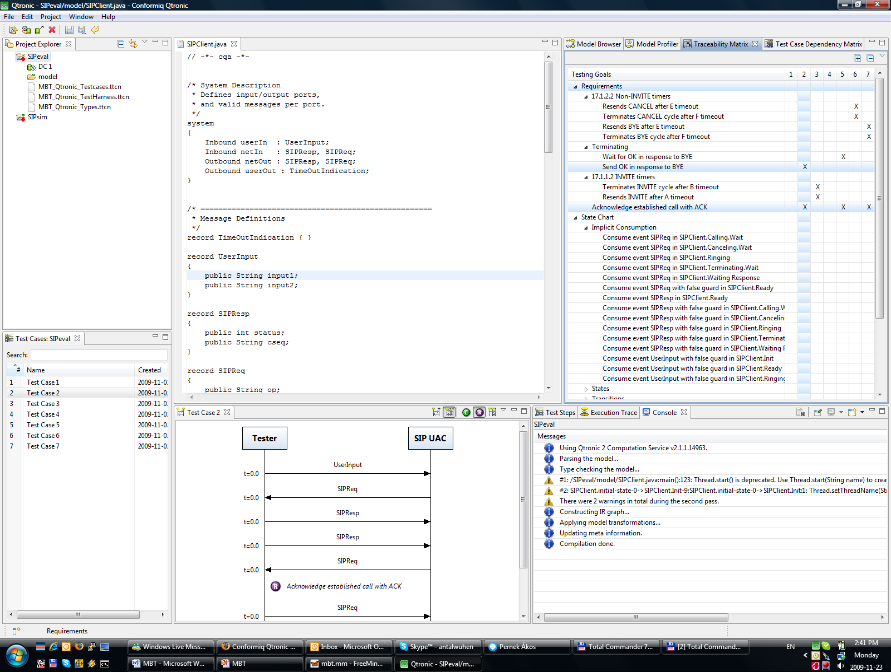


Figure 5 Qtronic perspective in Eclipse

Adaptation is done by plug-ins: scripter plug-in for offline while MBT and both adapter and logging plug-ins for online testing. The plug-ins can be performed in C++ or Java. The Qtronic package already has some plug-ins, for example TTCN-3 or HTML scripter. It is easy to make a new plug-in for a specific format.

Conformiq Qtronic is a true MBT tool with a very general approach. Open plugins makes the tool highly flexible and easily adaptable to different domains.

## Spec Explorer ‎[7]

Spec Explorer is a MBT tool that is allowed for use in any non-commercial purpose. It is made by Microsoft and accepts only Windows as its operating system. It can test in both offline and online approaches. Spec Explorer is strongly tied to Visual Studio. It uses Visual Studio’s (VS) formats, and does compilation in VS.

Spec Explorer uses the textual notations: Abstract State Machine Language (AsmL) and Spec# for modeling. ASML is an executable specification language based on the theory of Abstract State Machines. Spec# is an extended version of C#, with extension to support non-null types and checked exceptions. Modeling can be done with text editors or with an integrated graphical editor. Spec Explorer generates visual finite state machine (FSM) from textual notation for illustration.

When there is a requirement to run the test suite automatically against the implementation of the system it was necessary to write an adapter for mapping Spec Explorer and SUT together. The adapter may be written in C# or Visual Basic.

Spec Explorer offers few coverage criteria. The offline approach gives random walk, transition coverage or shortest path algorithm. Online testing works only with randomly walking. There are also some searching algorithms for sharpening test set quality that affect both testing approaches.

Both offline and online testing are executable in Spec Explorer. It is also possible to export offline test suite in xml format or export executable test code in Visual Basic or C# language. Online testing can be started directly from the Spec Explorer and the tool will continue to run test cases against the model until SUT fails or the user stops the execution process.

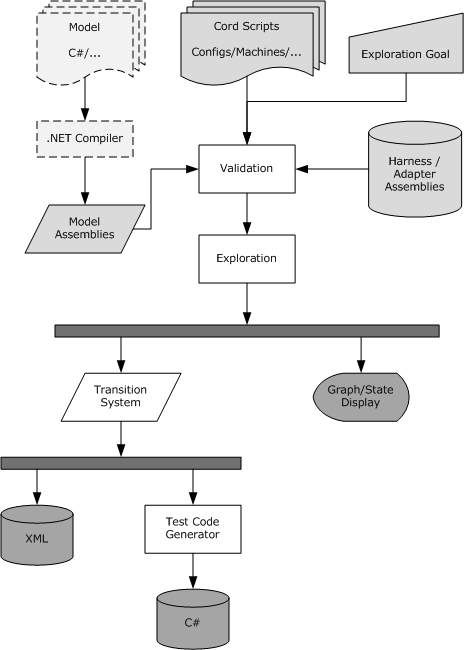


Figure 6 Workflow of Spec Explorer

Spec Explorer can receive the following inputs:

* A set of .NET model assemblies. The .NET assemblies are created from a model program in C# or another .NET language. Model source is compiled with a standard compiler. It is usually annotated with custom attributes for marking modeling constructs. Other model notations, such as diagrams, can be incorporated by first translating them into C# or to the Cord scripting language.
* A set of .NET harness, or adapter, assemblies. These can potentially include the implementation under test. If the implementation under test is included, its presence enables the tool to validate consistency of bindings from model to implementation. There is no difference between model assemblies and implementation/harness/adapter assemblies; that is, they all constitute a single set of assemblies referenced by Spec Explorer.
* A set of coordination (Cord) scripts, which describe action sets, mapping to metadata in provided assemblies, configuration options, and behaviors in the form of action machine definitions.
* An exploration goal provided through the user interface (UI) or the command line, which identifies the machine to explore and the exploration method to use for it.

Based on these inputs, the core algorithmic functionality of the tool is the following:

* Validating the consistency of the inputs. This means checking whether references in the scripts to elements in the assemblies are resolvable, as well as checking the syntax and context conditions of the script language itself.
* Running exploration, as defined by the exploration goal. This means systematically running the goal machine to determine the reachable state space and state transitions.

The goal machine can be specified by the scripts, by a reference to a model program from the assemblies, or by composition or transformation of other machines. Processes traditionally available in model-based testing tools such as traversals are represented as particular transformations on machines, which provides a uniform paradigm for state exploration and traversal.

The outputs from running an exploration are presented and processed in the following way:

* The state graph that results from exploration. The user can click states and steps in the graph and inspect their content and properties. The display can be configured in various ways in a Cord script.
* The graph is also encoded in a data structure called the transition system. This is a programmatic representation with a direct correspondence to an XML format that can be output. Moreover, the transition system can be translated into C# code that runs offline tests.

Because exploration and traversal are unified, the result of exploration—whether displayed as a graph, XML transition system, or C# test suite—can represent various logical artifacts: the explored state space, a test suite, a model-checking counterexample, or the log of a test run.

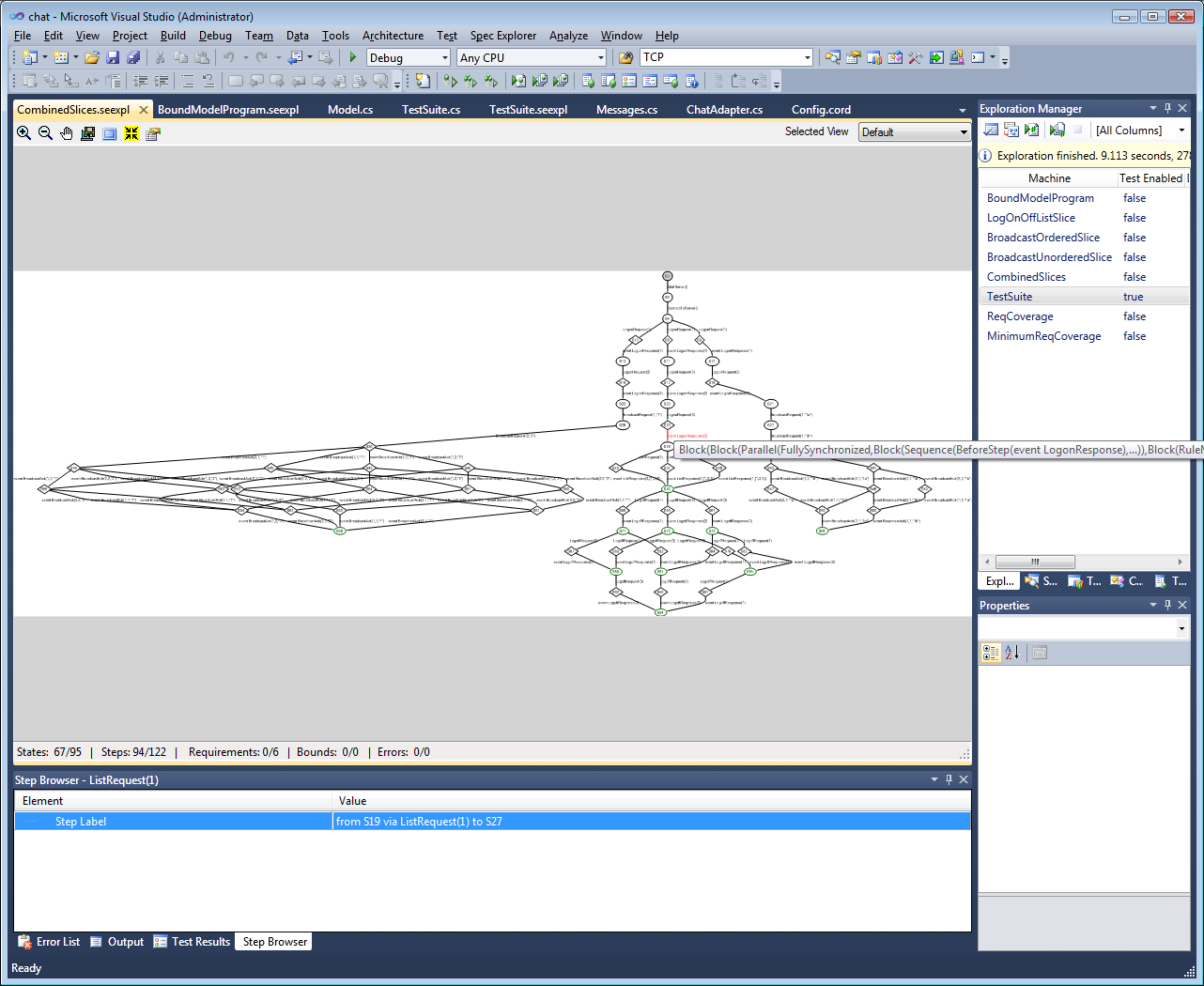


Figure 7 Spec Explorer in Visual Studio - Exploration

Spec Explorer is most useful when you are familiar with Visual Studio. Unfortunately, it is just for research purposes. Based on this, however, a very sophisticated tool from Microsoft is being developed.

# The MBT Application Library

## Overview

The architecture of a TitanSim based test harness can be seen in Figure 8. The applib provides an external “MBT Applib Interface” on which it is possible to control and communicate with the test harness. This interface carries primitives that make it possible to invoke and execute test steps of the application libraries. The events reported back by the applications are also sent on this interface back to the MSC Executor.

The applib can be extended so that a user defined port with user defined messages can be handled with the help of an applib. This “User mapping” code is responsible to map the incoming messages to either TitanSim functions/test steps or user defined functions. The applib events can be also caught and mapped to user defined messages, which can be sent back to the Tester component.



Figure 8 TitanSim based Test Harness

The MBT applib provides a simple FSM that can catch the reported events from the applibs and can execute all desired test steps registered by the applibs. The behavior of the FSM can be controlled via functions in the “User mapping” code, or via MBT Applib PDUs on the “MBT Applib Interface”.

The user glue code is where the applibs are initialized and put together. This part can also contain some user written support functions to further extend the test harness.

There are two main approaches to use this test harness arrangement:

1. The test harness can be controlled via the “MBT Applib Interface”. In this case, the Tester must communicate via MBT Applib PDUs, that is the model must be built so that it communicates with the environment using MBT applib PDUs.
2. The other alternative is that the model is using its own PDUs to interact with the environment. But then, the test harness must be extended with “User mapping” code that can handle these messages and can translate to MBT Applib primitives.

## Simple MBT Applib Demo

A simple TitanSim project demonstrating the usage of the MBT Applib can be found in the demo directory of the product. It consists of the following files:

* MBT\_demo.prj  
  mctr\_gui project file containing all information to be able to build the project.
* MBT\_demo.ttcn  
  TTCN module that contains all the TTCN sources that were written for this demo.
* MBT\_demo.cfg  
  Sample config file in order to be able to execute the demo
* Makefile\_patch.sh  
  Makefile patcher shell script that is used by mctr\_gui to build the demo project.

### Test arrangement



Figure 9 Component Arrangement in the simple demo

The component arrangement of the demo can be seen in Figure 9. In the testcase started on MTC the following steps are executed:

1. Creating the Tester Component
2. Creating and connecting with the Test Harness
3. The behaviors are started on both components
4. Waiting for shutdown

Exampel:

testcase tc\_MBT\_HTTP\_Demo() runs on MBT\_Demo\_Main\_CT

{

f\_EPTF\_Base\_init\_CT("mtc");

var MBT\_Demo\_LGen\_CT vc\_MBT\_CT := MBT\_Demo\_LGen\_CT.create;

var MBT\_Demo\_Tester\_CT vc\_Tester\_CT := MBT\_Demo\_Tester\_CT.create;

connect(vc\_MBT\_CT:EPTF\_MBT\_PCO, vc\_Tester\_CT:EPTF\_MBT\_PCO);

vc\_MBT\_CT.start(f\_MBT\_Demo\_LGen\_HTTP\_behavior());

vc\_Tester\_CT.start(f\_MBT\_Demo\_Tester\_HTTP\_behavior());

//all component.done;

f\_EPTF\_Base\_wait4Shutdown();

}

### Tester Behavior

The Tester component is an instance of the MBT\_Demo\_Tester\_CT component type which extends the EPTF\_MBT\_Tester\_CT component type defined by the MBT Applib. This component has an EPTF\_MBT\_PCO port, so it can communicate with the Test Harness using the MBT Applib’s API.

The main function which is executed on this component has the following structure:

1. Initialization
2. EPTF\_MBT\_ConfigRequest PDU is sent to the Test Harness to setup the entity group. The ConfigResponse must arrive to indicate, that the configuration was successful.
3. EPTF\_MBT\_TestStepRequests are sent to to execute protocol specific behavior in the test harness, and the incoming responses are reported back to the Tester component via EPTF\_MBT\_TestStepResponse PDUs
4. Finally, a quit command is sent to the Test Harness
5. The Tester waits until the clean up process is finished using the f\_EPTF\_Base\_wait4Shutdown() function.

Example:

function f\_MBT\_Demo\_Tester\_HTTP\_behavior() runs on MBT\_Demo\_Tester\_CT

{

f\_EPTF\_Base\_init\_CT("MBT\_Demo\_Tester");

EPTF\_MBT\_PCO.send(EPTF\_MBT\_ConfigRequest:

{

entityGroupName := "MBT\_EntityType",

noEntities := 1,

behaviors := {"MBT\_behavior", "HTTP Behavior"},

fsmName := "FSM\_MBT"

}

);

// TODO: activate default altstep, with timeout handling

EPTF\_MBT\_PCO.receive(EPTF\_MBT\_ConfigResponse:?);

EPTF\_MBT\_PCO.send(EPTF\_MBT\_TestStepRequest:

{"EPTF HTTP: Sends HTTP Request", {}, omit});

EPTF\_MBT\_PCO.receive(EPTF\_MBT\_TestStepResponse:

{ bName := "HTTP Behavior",

iName := "HTTP CHAR Response Message Received"});

EPTF\_MBT\_PCO.send(EPTF\_MBT\_CommandRequest:{ quit := {}});

f\_EPTF\_Base\_wait4Shutdown();

}

### Test Harness Behavior

The Test Harness is an instance of MBT\_Demo\_LGen\_CT which extends EPTF\_MBT\_LGen\_CT component type defined by the MBT applib and all required component types in order to be able to use the desired application libraries.

The main function on the Test Harness has the following responsibilities:

1. Init the component
2. Init the used application libraries
3. Init the MBT FSM
   1. Specify the function that will fill in the simulated entities user databases
4. The Test Harness waits until the clean up process is finished using the f\_EPTF\_Base\_wait4Shutdown() function.

Example:

function f\_MBT\_Demo\_LGen\_HTTP\_behavior() runs on MBT\_Demo\_LGen\_CT

{

f\_EPTF\_MBT\_init("MBT\_Demo\_LGen",0, "MBT\_");

f\_EPTF\_HTTP\_init(

pl\_selfName:="HTTP\_LoadGen",

pl\_selfId:=0,

pl\_entityNamePrefix:="HTTP\_AppLib\_",

pl\_numEntities:=0)

f\_EPTF\_HTTP\_LocalTransport\_init\_CT(

pl\_selfName:="EPTF\_HTTP\_LocalTransport",

pl\_receiveFunction:=refers(f\_EPTF\_HTTP\_messageReceived),

pl\_eventReceiveFunction:=refers(f\_EPTF\_HTTP\_eventReceived));

// Further HTTP applib initialization comes here (e.g. local transport)

f\_EPTF\_MBT\_initLGenFsm

(

null,

refers(f\_EPTF\_MBT\_HTTP\_Demo\_fillInDB)

);

f\_EPTF\_Base\_wait4Shutdown();

}

### Testcase Execution Example

An example scenario of the execution can be seen in Figure 10.



Figure 10 Simple Demo Signaling

1. Tester component is created, initialized and the corresponding behavior function is started.
2. Test Harness component is created, initialized and the corresponding behavior function is started.
3. EPTF\_MBT\_ConfigRequest is sent to create the entity group and activate the MBT\_FSM.
4. EPTF\_MBT\_ConfigResponse is sent to report the configuration was successful.
5. EPTF\_MBT\_TestStepRequest is sent to execute the HTTP send request in the Test Harness
6. HTTP Request is sent out to the SUT.
7. The SUT responds with an answer.
8. The HTTP applib reports an event to the FSM, which in turn generates an EPTF\_MBT\_TestStepResponse to the Tester. This PDU describes the event it received.
9. Finally an EPTF\_MBT\_CommandRequest is sent to exit from the current configuration and clean up the test system.

## MBT Applib Demo with Qtronic using User Defined Messages

The Conformiq Qtronic MBT tool is shipped with a demo that models a SIP phone. The goal of the MBT Applib demo for Qtronic is to implement a test harness that is able to generate real-life SIP messages for the abstract test generated from the SIP demo model.

### SUT: SIP Phone

To create the demo, the System under test was also implemented in the TitanSim Framework, using the MBT Applib. The SUT has two interfaces, the upper one can receive commands:

* Invite  
  To start a session
* Cancel  
  To cancel an initiated session
* Bye  
  To finish an established session

The lower interface:

* Can receive SIP messages (e.g. 200 OK, 486 Busy …)
* And send SIP messages (e.g. INVITE, CANCEL)

The state machine of the SIP phone can be seen in Figure 11. “i: “ mark means that the transition will be traversed when that message event is received in the given state, while “o: “ means, that during the state change those messages are sent which are enumerated there.



Figure 11 SUT: SIP Phone

The source code of the SUT implementation can be found in these module:

* demo/sut/MBT\_SUT\_SIP\_Phone.ttcn

### Demo Files

The demo consists of the following files:

* Qtronic Model of the SUT
  + demo/mapped/model/SIPClient.java
  + demo/mapped/model/SIPClient.xmi
* User written files
  + Demo/MBT\_Qtronic\_Demo.prj  
    Project file that builds the whole project.
  + Demo/MBT\_Qtronic\_Demo.cfg  
    Sample config file to be able to execute the testcases.
  + Demo/MBT\_Qtronic\_Demo.ttcn  
    User written code for initialization, configuration and mapping.
* Generated files by Qtronic
  + Demo/MBT\_Qtronic\_Testcases.ttcn  
    This contains the generated testcases.
  + Demo/MBT\_Qtronic\_TestHarness.ttcn  
    The send/receive functions can be found here.
  + Demo/MBT\_Qtronic\_Types.ttcn  
    The generated type definitions for the abstract test data.
* TTCN-3 Scripter
  + Demo/TTCNScripter/TTCNScripter.jar

### Test Arrangement Overview

The demo’s test arrangement can be found in Figure 12. There are three main actors: the Tester, which executes the abstract test; the Test Harness, which provides the mapping between the abstract test data and a real-life test PDU; and finally the System Under Test.



Figure 12 Qtronic Demo Component Structure

The Tester controls the SUT and the Test Harness. It is built mostly from generated code, the only exceptions are the function that are responsible for creating and connecting the components.

The Test Harness contains the MBT and the SIP application libraries. On top of these is the user mapping code, which must implement the mapping between the MBT Applib/SIP Applib API and the abstract test data.

### Tester Code Examples

The Tester component is an instance of the MBT\_Qtronic\_Demo\_Tester\_CT component type which extends the EPTF\_MBT\_Tester\_CT component type defined by the MBT Applib. This component has an EPTF\_MBT\_PCO port, so it can communicate with the Test Harness using the MBT Applib’s API.

The f\_MBT\_Qtronic\_Demo\_beginTestcase() function on the Test Harness is invoked each time a test case is started. It’s role is to create, connect and init the components:

1. Initialization
2. Create the Test Harness
3. Connect the Test Harness with the Tester
4. EPTF\_MBT\_ConfigRequest PDU is sent to the Test Harness to setup the entity group. The ConfigResponse must arrive to indicate, that the configuration was successful.

Example:

function f\_MBT\_Qtronic\_Demo\_beginTestCase() runs on Qtronic\_CT

{

log(%definitionId, " started");

f\_EPTF\_Base\_init\_CT("mtc");

log(%definitionId, " Creating LGen");

var MBT\_Qtronic\_Demo\_LGen\_CT vc\_lgen := MBT\_Qtronic\_Demo\_LGen\_CT.create;

connect(self:netIn, vc\_lgen:netOut);

connect(self:netOut, vc\_lgen:netIn);

connect(self:EPTF\_MBT\_PCO,vc\_lgen:EPTF\_MBT\_PCO);

vc\_lgen.start(f\_MBT\_Qtronic\_Demo\_LGen\_behavior());

EPTF\_MBT\_PCO.receive(EPTF\_MBT\_CommandResponse:?) from vc\_lgen;

EPTF\_MBT\_PCO.send(EPTF\_MBT\_ConfigRequest:

{

entityGroupName := "MBT\_EntityType",

noEntities := 1,

behaviors := {"MBT\_behavior", "Behavior\_SIP"},

fsmName := "FSM\_MBT"

}

) to vc\_lgen;

EPTF\_MBT\_PCO.receive(EPTF\_MBT\_ConfigResponse:?) from vc\_lgen;

log(%definitionId, " LGen ready");

log(%definitionId, " finished");

}

The f\_MBT\_Qtronic\_Demo\_endTestcase() function on the Test Harness is invoked each time a test case is started. It’s role is to create, connect and init the components:

1. Stop all the components
2. Wait until all components are finished with the clean up.

Example:

function f\_MBT\_Qtronic\_Demo\_endTestCase() runs on Qtronic\_CT

{

log("### MAIN: END TESTCASE started");

f\_EPTF\_Base\_stopAll(none, true);

log("### MAIN: END TESTCASE finished");

}

### Test Harness Code Examples

#### Test Harness Behavior function

The Test Harness is an instance of MBT\_Qtronic\_Demo\_LGen\_CT which extends EPTF\_MBT\_LGen\_CT component type defined by the MBT applib and all required component types in order to be able to use the desired application libraries.

The main function on the Test Harness has the following responsibilities:

1. Init the component
2. Init the used application libraries
3. Init the MBT FSM
   1. Specify the function that will fill in the simulated entities user databases
4. Activate the altstep that will handle the incoming abstract test data PDUs.
5. The Test Harness waits until the clean up process is finished using the f\_EPTF\_Base\_wait4Shutdown() function.

Example:

function f\_MBT\_Qtronic\_Demo\_LGen\_behavior() runs on MBT\_Qtronic\_Demo\_LGen\_CT

{

activate(as\_MBT\_Qtronic\_Demo\_LGen\_userMessageHandler());

f\_EPTF\_MBT\_init("MBT\_Demo\_LGen",0, "MBT\_")

f\_SIP\_applibInit("MBT\_Demo\_LGen");

f\_EPTF\_SIP\_LocalTransport\_init(tsp\_MBT\_SIP\_Transport);

vf\_EPTF\_SIP\_LocalTransport\_receive :=

refers(f\_EPTF\_SIP\_Message\_MsgHandler);

vf\_EPTF\_SIP\_LGen\_msgSender :=

refers(f\_EPTF\_SIP\_LocalTransport\_sendSIPMessage);

v\_removeUAS :=

refers(fcb\_EPTF\_SIP\_LocalTransport\_removeUAS);

f\_EPTF\_MBT\_initLGenFsm

(

refers(f\_MBT\_Qtronic\_Demo\_LGen\_createUserMessage),

refers(f\_MBT\_Qtronic\_Demo\_LGen\_fillInDB)

);

EPTF\_MBT\_PCO.send(EPTF\_MBT\_CommandResponse:{ ready := {}}) to mtc;

f\_EPTF\_Base\_wait4Shutdown();

}

#### User mapping

The mapping of the abstract test data used in the model is realized in two functions:

* An altstep must be written and activated which is able to process the incoming abstract test data and map it to Applib functions.
* A function must be written, which is able to produce abstract test data based on TitanSim Applib events, or real life PDUs.

Example Altstep:

altstep as\_MBT\_Qtronic\_Demo\_LGen\_userMessageHandler()

runs on MBT\_Qtronic\_Demo\_LGen\_CT

{

var SIPResp vl\_SIPResp;

var SIPReq vl\_SIPReq;

var EPTF\_LGenBase\_TestStepArgs vl\_stepArgs :=

{ eIdx := 0, refContext :=

{ fCtxIdx := 0, fRefArgs := { } }, stepArgs := { }

};

[] netOut.receive(SIPResp:?) -> value vl\_SIPResp

{

f\_EPTF\_SchedulerComp\_refreshSnapshotTime();

log(%definitionId & "(): incoming ", vl\_SIPResp);

if (vl\_SIPResp.status == 180)

{

vl\_stepArgs.refContext.fRefArgs := {c\_status\_180Ringing\_idx};

f\_SIP\_step\_createResponse(vl\_stepArgs);

}

else if (vl\_SIPResp.status == 200)

{

if (vl\_SIPResp.cseq == "INVITE")

{

vl\_stepArgs.refContext.fRefArgs :=

{c\_status\_200OK\_idx, c\_SIP\_Method\_INVITE};

f\_SIP\_step\_createResponse(vl\_stepArgs);

}

else if (vl\_SIPResp.cseq == "CANCEL")

{

vl\_stepArgs.refContext.fRefArgs :=

{c\_status\_200OK\_idx, c\_SIP\_Method\_CANCEL};

f\_SIP\_step\_createResponse(vl\_stepArgs);

}

}

//. . .

else

{

log(%definitionId & "(): unhandled SIPResp");

}

repeat;

}

[] netOut.receive(SIPReq:?) -> value vl\_SIPReq

{

f\_EPTF\_SchedulerComp\_refreshSnapshotTime();

log(%definitionId & "(): incoming ", vl\_SIPReq);

if (vl\_SIPReq.op == "BYE")

{

f\_SIP\_step\_createBYE(vl\_stepArgs);

}

else

{

log(%definitionId & "(): unhandled SIPReq");

}

repeat;

}

}

Example Abstract PDU Producer:

function f\_MBT\_Qtronic\_Demo\_LGen\_createUserMessage

(in EPTF\_LGenBase\_TestStepArgs pl\_ptr)

runs on MBT\_Qtronic\_Demo\_LGen\_CT

{

var charstring vl\_param := "sip:127.0.0.1:5061";

if (pl\_ptr.reportedEvent.event.bIdx == v\_SIP\_myBIdx and

pl\_ptr.reportedEvent.event.iIdx == c\_SIP\_eventIdx\_INVITE)

{

var integer vl\_FSMIdx := -1;

pl\_ptr.refContext.fCtxIdx := 0;

if (not f\_EPTF\_SIP\_FSMInitialized(pl\_ptr.eIdx,

pl\_ptr.refContext.fCtxIdx, vl\_FSMIdx))

{

f\_SIP\_step\_init(pl\_ptr);

f\_SIP\_step\_handleINVITE(pl\_ptr);

}

netIn.send(SIPReq:{"INVITE", vl\_param});

}

else if (pl\_ptr.reportedEvent.event.bIdx == v\_SIP\_myBIdx

and pl\_ptr.reportedEvent.event.iIdx == c\_SIP\_eventIdx\_ACK)

{

netIn.send(SIPReq:{"ACK", vl\_param});

}

else if (pl\_ptr.reportedEvent.event.bIdx == v\_SIP\_myBIdx and

pl\_ptr.reportedEvent.event.iIdx == c\_SIP\_eventIdx\_BYE)

{

netIn.send(SIPReq:{"BYE", vl\_param});

}

else if (pl\_ptr.reportedEvent.event.bIdx == v\_SIP\_myBIdx and

pl\_ptr.reportedEvent.event.iIdx == c\_SIP\_eventIdx\_CANCEL)

{

netIn.send(SIPReq:{"CANCEL", vl\_param});

}

// . . .

else

{

log(%definitionId & "(): unhandled incoming message");

}

}

### Workflow

The work process can be outlined in the following steps:

* Modeling
  + Identify the interfaces of the model
  + Create the model in Qtronic
* Test Harness Implementation
  + Create the Test Harness component
    - Put all required component types together
    - Write the initialization code
    - Plan the mappings between the Model interface and the Applibs’ interfaces
    - Write the mapping code
  + Write the configuration functions
    - beginTestCase
    - endTestCase
* Build the Executable Test Suite
  + Export the TTCN-3 code generated with the TTCN-Scripter
  + Create the project file with all TTCN sources
  + Build the project
* Test Execution
  + Write the TITAN configuration file
  + Execute the Test
* Analyze the test results.

### TTCNScripter settings

The settings for the TTCNScripter Qtronic plugin used for this demo can be seen in the following figures:

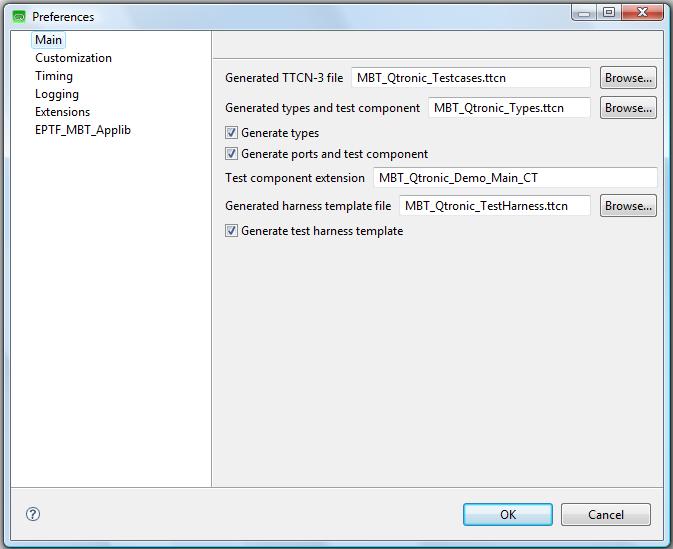


Figure 13 TTCNScripter: Main Panel

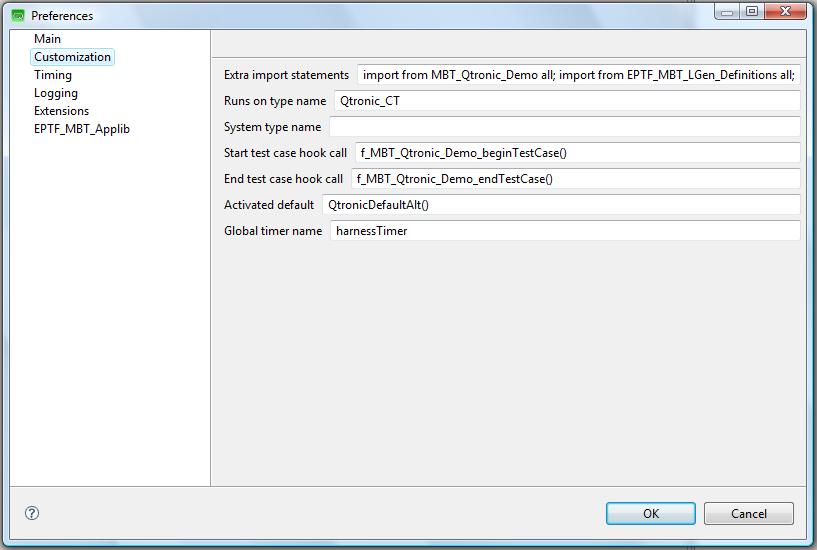


Figure 14: TTCNScripter: Customization Panel

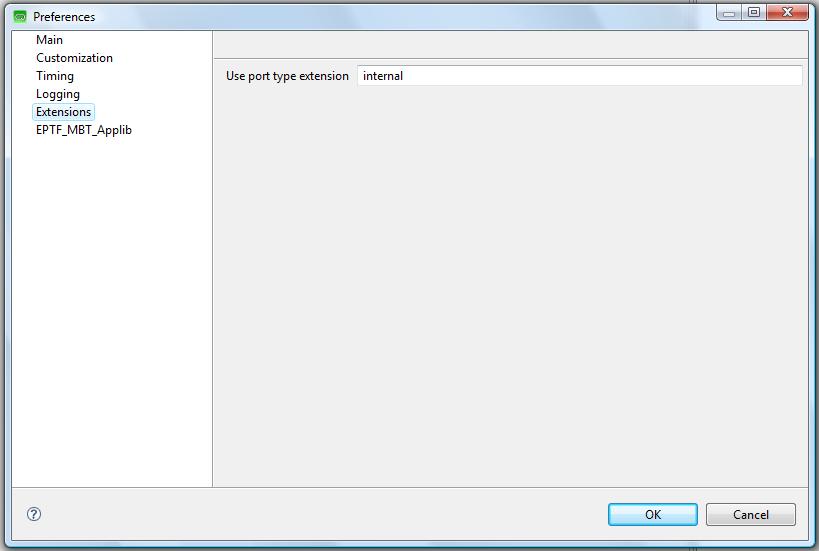


Figure 15 TTCNScripter: Extensions Panel

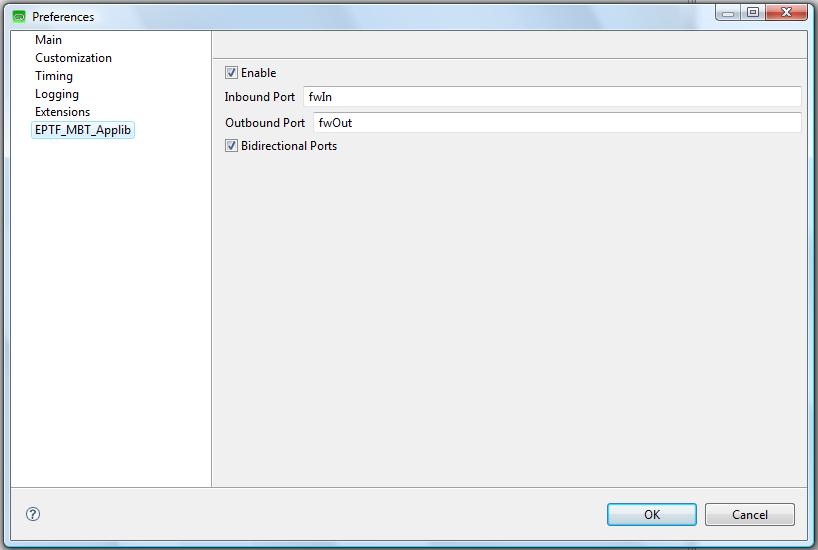


Figure 16 TTCNScripter: EPTF\_MBT\_Applib Panel

## MBT Applib Demo with Qtronic using Applib PDUs

In this demo we modify the SIP Phone model that is shipped with Qtronic and was used in the previous example (see ‎4.3.1) so that it will use TitanSim test steps and events directly. Therefore, no user mapping code is needed.

### Demo Files

The demo consists of the following files:

* Qtronic Model of the SUT
  + The SUT Behavior
    - demo/direct/model/SIPClient.java
    - demo/direct/model/SIPClient.xmi
  + Java definitions of the TitanSim API:
    - src/Qtronic/model/EPTF\_MBT\_Framework.java
    - src/Qtronic/model/EPTF\_MBT\_SIP\_Applib.java
* User written files
  + Demo/direct/MBT\_Qtronic\_Demo.prj  
    Project file that build the whole project.
  + Demo/direct/MBT\_Qtronic\_Demo.cfg  
    Sample config file to be able to execute the testcases.
  + Demo/direct/MBT\_Qtronic\_Demo.ttcn  
    User written code for initialization, configuration and mapping.
* Generated files by Qtronic
  + Demo/direct/MBT\_Qtronic\_Testcases.ttcn  
    This contains the generated testcases.
  + Demo/direct/MBT\_Qtronic\_TestHarness.ttcn  
    The send/receive functions can be found here.
  + Demo/direct/MBT\_Qtronic\_Types.ttcn  
    The generated type definitions for the abstract test data.
* TTCN-3 Scripter
  + Demo/TTCNScripter/TTCNScripter.jar

### Test Arrangement Overview

The demo’s overview picture can be found in Figure 17. There are three main actors: the Tester, which executes the abstract test; the Test Harness, which provides the mapping between the abstract test data and a real-life test PDU; and finally the System Under Test.

The notable difference between this and the mapped demo is that the user doesn’t need to implement the “mapping” layer in the Test Harness. Instead, some TitanSim API definitions are included in the model, and the model communicates with only those abstract data definitions, that are allowed there.



Figure 17 Qtronic Direct Demo Overview

Consequently everything described in the previous section (see ‎4.3) applies here as well, except for the mapping functions, because they are not needed now.

### Modeling with TitanSim API

The Qtronic model must be modified, so that it can communicate with MBT Applib primitives:

system

{

Inbound userIn : UserInput;

Outbound userOut : TimeOutIndication;

**Inbound fwIn : EPTF\_MBT\_TestStepRequest;**

**Outbound fwOut : EPTF\_MBT\_TestStepResponse;**

}

Message sending in the model is done via separate functions. For example in Figure 18 there is a transition from the Init state to the Calling state. When this transition is executed it calls the *Invite()* model function, which will send out a SIP INVITE abstract test message.

The *Invite()* function will send out an EPTF MBT Applib *TestStepResponse* primitive to the model environment. During test generation it will be mapped to a message receive statement, where the Tester expects a *TestStepResponse* message from the Test Harness:

public void Invite() {

EPTF\_MBT\_TestStepResponse r;

r.bName = c\_SIP\_Behavior;

r.iName = c\_SIP\_eventName\_INVITE;

fwOut.send(r, 1.0); }

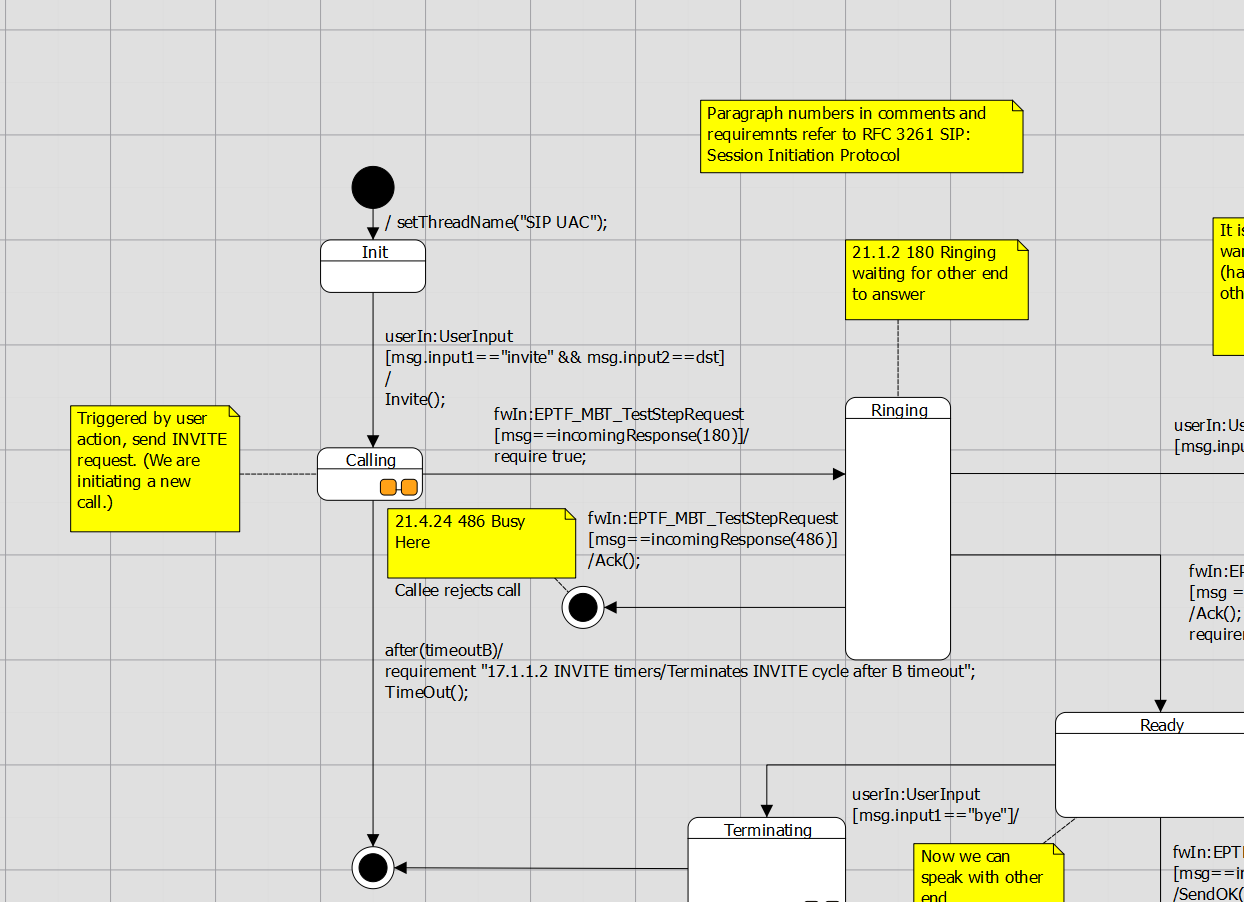


Figure 18 Model fragment in Qtronic with the direct approach

Message reception is also implemented using CQL functions in the model. When we have an incoming 180 response in the Calling state we will go to the Ringing state:

fwIn: EPTF\_MBT\_TestStepRequest

[msg == incomingResponse(180)]/

require true;

The *incomingResponse()* model function is using the *EPTF\_MBT\_TestStepRequest* MBT Applib primitive to call a *createResponse* SIP Applib test step:

public EPTF\_MBT\_TestStepRequest incomingResponse(int p\_answerCode)

{

EPTF\_MBT\_TestStepRequest r;

r.stepName = c\_SIP\_stepFunction\_createResponse;

r.stepArgs =

{

f\_EPTF\_SIP\_mapAnswerCode2SipTemplateCode(p\_answerCode)

};

r.addr.entityGroupName = "MBT\_EntityType";

r.addr.eIdx = 0;

r.addr.fIdx = 0;

return r;

}

With this modeling approach the generated testcases will always use the EPTF MBT Applib primitives to communicate with the test harness.

## Demo with Spec Explorer using User Defined Messages

The goal of this demo is to create a project in Spec Explorer which is able to generate a functionally same code as Qtronic does. Thus the operation and structure are not detailed in this passage due to similarity.

### Generating TTCN code in Spec Explorer

Spec Explorer originally does not support TTCN code generation. So, some additional classes are needed to do that. The operation in short is that test cases generated by Spec Explorer invoke the methods of these classes, so during the running they indirectly write TTCN code in a separate file. The correctness of this TTCN code is the tester’s responsibility. Spec Explorer checks the success of its own test cases only, the generated ones can be checked with TITAN. This whole procedure looks like as a double test case generation, one for Spec Explorer and one for TITAN.

### Export2TTCN.cs

Export2TTCN.cs contains the necessary classes to export TTCN code from Spec Explorer. It implements the Export2TTCN namespace, which contains two classes:

Figure 19 Class FileExport

Class FileExport

-fileBuffer : string[]

-rowNumber : int

-filename : string

-c\_indent : int

-act\_indent : int

+FileExport(filename)

+ind\_p() : void

+ind\_n() : void

-getindented(row : string) : string

+write(row : string) : void

+dump() : void

+write\_tc(row : string) : void

+dump\_tc() : void

This is a universal class for writing files with indentation handling (ind\_p increases, ind\_n decreases indent). Test cases must be written by 2 specialized methods: write\_tc and dump\_tc.

TTCNExporter is an abstract class for writing TTCN files. It has 6 abstract methods to be configurable:

Figure 20 Class TTCNExporter

*Class TTCNExporter*

-file : FileExport

-started : bool

-nof\_testcases : int

-modulename : string

-filename : string

#TTCNExporter(p\_modulename : string, p\_filename : string)

+is\_started() : bool

*#write\_imports() : void*

*#write\_types() : void*

*#write\_altsteps() : void*

*#write\_else() : void*

*#beginning\_tc() : void*

*#ending\_tc() : void*

-start\_testcase() : void

+start() : void

+end() : void

#write(row : string) : void

+write\_tc(row : string) : void

+ind\_p() : void

+ind\_n() : void

### The procedure of creating a new project

* File/new/project –> choose Visual C#/Test/Spec Explorer Base Solution  
  Set parameters (name, path etc.)  
  Choose Static modeling solution
* Implement Adapter.cs using prepared class(es) (TTCNExporter)  
  Override abstract methods
* Implement Model.cs  
  Declare necessary actions
* Implement Config.coord  
  Declare actions and switches in the config Main  
  Create machines, especially the three main machines (construct model program, synchronizing, construct test cases)
* In the Exploration Manager select Testsuite machine (which creates test cases) and click on Generate Test Code
* On the Test Tools Toolbar click on the “Run All Tests in Solution

### FSM implementation techniques

Figure 21 FSM 1

This simple figure shows three states and two transitions. S1 is a start state, the others are accepting states.

Spec Explorer does not support drawing models, so it has to be written as a program. First of all create a machine for each state:

Machine S1() : ConfigName

{}

Machine S2() : ConfigName

{}

Machine S3() : ConfigName

{}

The body of machines contains the outgoing transitions, which can be accomplished as pairs of actions and states. For example (T1() ; S2()) | (T2() ; S3()). Semicolon means sequence, and pipe means “or”. Thus the next step from S1 is to execute T1 and go on into S2 or T2 and go on into S3. The final code of S1:

Machine S1() : ConfigName

{ (T1() ; S2()) | (T2() ; S3()) }

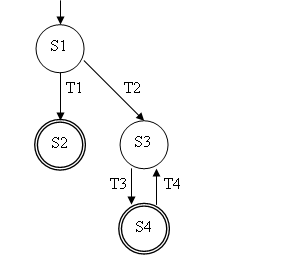


Figure 22 FSM 2

This figure is a bit more complicated. There is a cycle in the model. Spec Explorer does not support cyclic references between machines so we have to use something else. The problem can be treated with using \* operator, for example A\* means: repeats A zero or more times. One solution:

Machine S1() : ConfigName

{ (T1() ; S2()) | (T2() ; (T3() ; T4())\* ; S3()) }

Machine S2() : ConfigName

{}

Machine S3() : ConfigName

{ T3() ; S4() }

Machine S4() : ConfigName

{}

### Demo files

The project consists of three main files: Adapter.cs, Model.cs, Config.cord.

Adapter.cs contains two classes: NewExporter and Adapter. NewExporter is the real exporter class, it is inherited from TTCNExporter whose abstract methods are implemented by overriding them. Adapter is a static class, its methods are the implementation of transitions from FSM. Example:

using System;

using System.Collections.Generic;

using System.Text;

using Export2TTCN;

namespace MBT.Adapter

{

public class NewExporter : TTCNExporter

{

public NewExporter(string p\_modulename,string p\_path)

:base(p\_modulename,p\_path) {}

protected override void write\_imports() {…}

protected override void write\_types() {…}

protected override void write\_altsteps() {…}

protected override void write\_else() {…}

public void send(string command, string message)

{ write\_tc(command + "(" + message + ");");}

public void receive(string command, string message, double wait)

{write\_tc("-timername-.start(" + wait.ToString("F2") + " + SLACK);");

write\_tc("log(\"Waiting for: " + message + "\");");

write\_tc(command + "(" + message + ");");

write\_tc("-timername-.stop;");

}

protected override void beginning\_tc() {…}

protected override void ending\_tc() {…}

}

public static class Adapter

{

private static NewExporter exporter =

new NewExporter("MBT\_SpecExpl\_Testcases", "C:/path..");

public static void start()

{ exporter.start(); }

public static void end()

{ exporter.end(); }

public static void Transition\_name() {…send,receive…}

}}

Model.cs contains only one class: ModelProgram. The model can be programmed by this class. For example conditions can be set at transitions (actions). In this demo there was no need to use more than an empty program. Example:

using System;

using System.Collections.Generic;

using System.Text;

using System.Linq;

using Microsoft.Modeling;

namespace MBT.Model

{

static class ModelProgram

{

[Action]

static void start() {}

[Action]

static void Init\_Calling() {}

[Action]

static void Calling\_Ringing() {}

[Action]

static void Terminating\_OK() {}

[Action]

static void Terminating\_Timeout() {}

[Action]

static void end() {}

}

}

Config.cord can be divided into two main passages: configurations and machines. Configurations are used to control exploration and test generation. One of their purposes is to define the actions on which the model is based. This set of actions represents steps in a model trace or in the execution of an implementation. A configuration can also include a set of switches and parameters to control exploration and testing. Example:

using MBT.Adapter;

config Main

{

action abstract static void Adapter.start();

action abstract static void Adapter.end();

action abstract static void Adapter.Init\_Calling();

switch StepBound = 1024;

switch PathDepthBound = 1024;

switch StateBound = 1024;

switch TestClassBase = "vs";

switch GeneratedTestPath = "..\\TestSuite";

switch GeneratedTestNamespace = "MBT.TestSuite";

switch TestEnabled = false;

switch ForExploration = false;

}

machine Model() : Main where ForExploration = true

{ Model\_CLOSED\_START()\* }

Instead of states and transitions, FSM is modeled by machines. A machine can invoke another one or an action. So in every machine a pair of action and machine means a transition to a new state.

There are three main machines that must not be missed:

machine ModelProgram() : Main

{ construct model program from Main

where namespace = "MBT.Model" }

This construct builds a behaviour from a model program. The identifier is a required configuration name, which defines the actions to which the model program is bound.

machine Model\_Sync() : Main where ForExploration = true

{Model || ModelProgram}

This machine synchronizes Model and ModelProgram.

machine Testsuite() : Main where ForExploration = true, TestEnabled = true

{ construct test cases

where AllowUndeterminedCoverage = true

for Model\_Sync }

This machine creates the test cases.

### Exploration

Exploration is an operation by which Spec Explorer systematically discovers all possible states defined by a model, and the steps to transition from one state to another. The unit of exploration in Spec Explorer is a machine. Machines are defined in Cord scripts and each machine is based on one or more configurations. Configurations declare the set of actions that can be used to lebel steps, together with switches that control the exploration process. The result of an exploration is an Exploration Graph, which is displayed in the Exploration Graph Viewer.

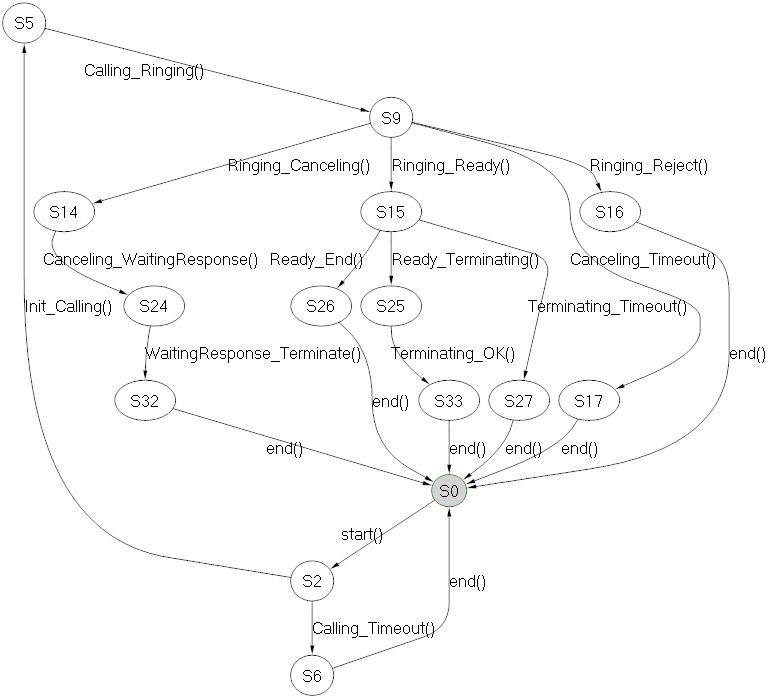


Figure 23 Exploration of Model machine

Exploration of Model machine shows the whole model. This is a mass of cycles starting with start() method and always ending with end(). These two methods are necessary to initialize and close each test case.

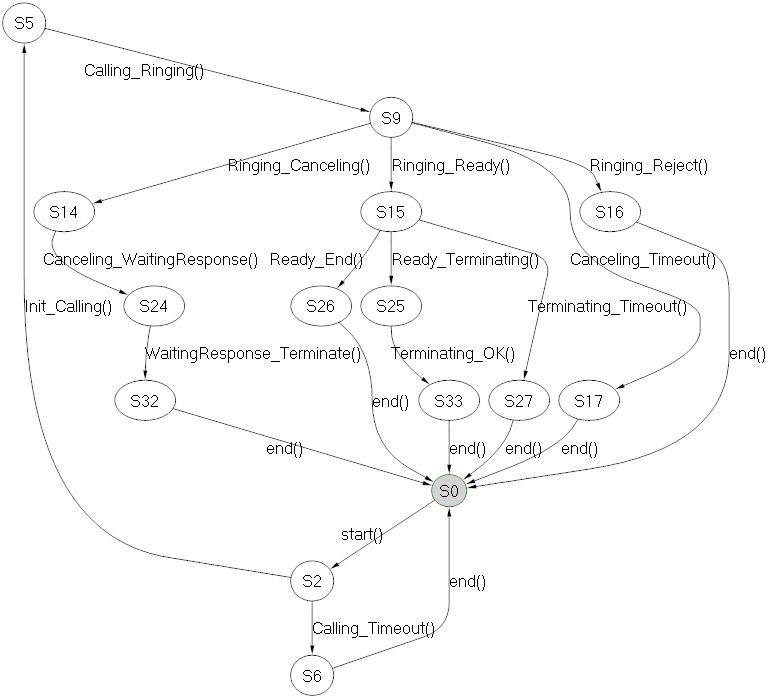


Figure 24 Exploration of Model\_Sync machine

This is the same as the first one, because ModelProgram in Model.cs is only a skeleton, actually it is empty.

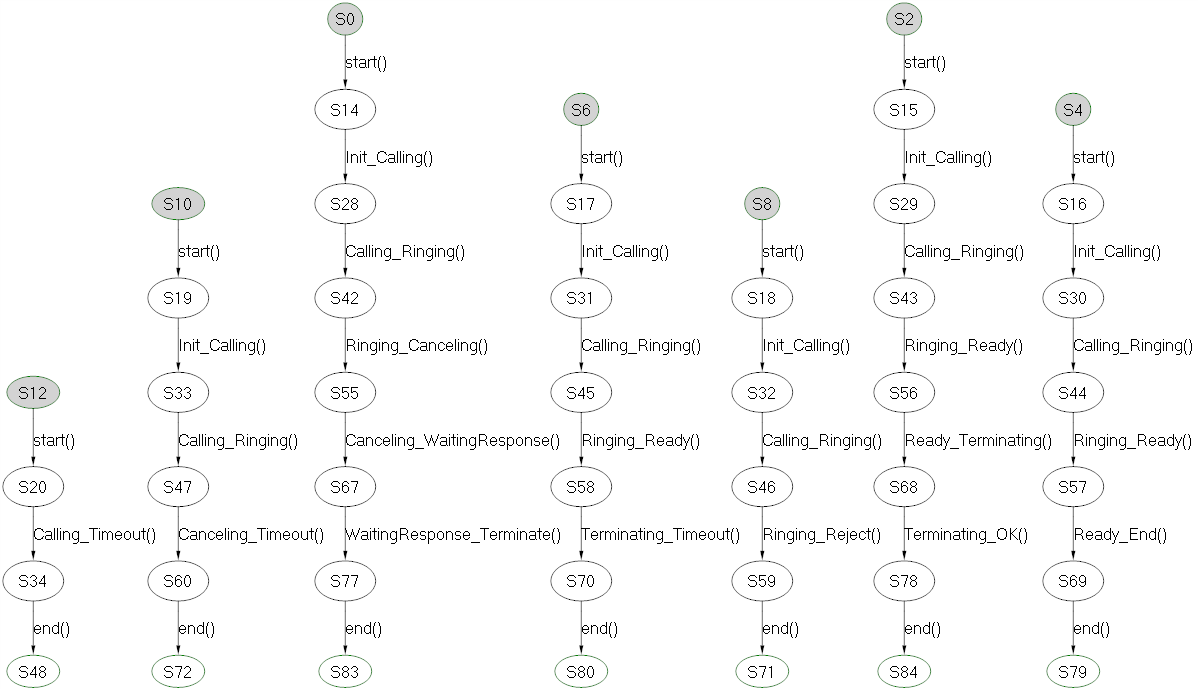


Figure 25 Exploration of Testsuite machine

Testsuite machine generates test cases from the model. The figure shows seven generated test cases with states and actions.